Construct, Merge, Solve and Adapt versus Large Neighborhood Search for Solving the Multi-Dimensional Knapsack Problem: Which One Works Better When? *

Evelia Lizárraga[1], María J. Blesa[1], and Christian Blum[2]

[1] Computer Science Deptartment, Universitat Politècnica de Catalunya – BarcelonaTech, Barcelona, Spain
{mjblesa, evelial}@cs.upc.edu
[2] Artificial Intelligence Research Institute (IIIA-CSIC)
Campus UAB, Bellaterra, Spain
christian.blum@iiia.csic.es

Abstract. Both, Construct, Merge Solve and Adapt (CMSA) and Large Neighborhood Search (LNS), are hybrid algorithms that are based on iteratively solving sub-instances of the original problem instances, if possible, to optimality. This is done by reducing the search space of the tackled problem instance in algorithm-specific ways which differ from one technique to the other. In this paper we provide first experimental evidence for the intuition that, conditioned by the way in which the search space is reduced, LNS should generally work better than CMSA in the context of problems in which solutions are rather large, and the opposite is the case for problems in which solutions are rather small. The size of a solution is hereby measured by the number of components of which the solution is composed, in comparison to the total number of solution components. Experiments are conducted in the context of the multi-dimensional knapsack problem.

1 Introduction

The development and the application of hybrid metaheuristics has enjoyed an increasing popularity in recent years [1, 2]. This is because these techniques often allow to combine the strengths of different ways of solving optimization problems in a single algorithm. Especially the combination of heuristic search with exact techniques—a field of research often labelled as matheuristics [3]—has been quite fruitful. One of the most well known, and generally applicable, algorithms from this field is called Large Neighborhood Search (LNS) [4], which is based on the

* This work was funded by project TIN2012-37930-C02-02 (Spanish Ministry for Economy and Competitiveness, FEDER funds from the European Union) and project SGR 2014-1034 (AGAUR, Generalitat de Catalunya). Evelia Lizárraga acknowledges funding from the Mexican National Council for Science and Technology (CONACYT, doctoral grant number 253787).

Lizárraga, E., Blesa, M., Blum, C. Construct, merge, solve and adapt versus large neighborhood search for solving the multi-dimensional Knapsack problem: Which one works better when?. A: European Conference on Evolutionary Computation in Combinatorial Optimization. "Evolutionary Computation in Combinatorial Optimization: 17th European Conference, EvoCOP 2017: Amsterdam, The Netherlands, April 19-21, 2017: proceedings". Springer, 2017, p. 60-74.
The final authenticated version is available online at https://doi.org/10.1007/978-3-319-55453-2_5

following general idea. Given a valid solution to the tackled problem instance, first, destroy selected parts of it, resulting in a partial solution. Then apply some other, possibly exact, technique to find the best valid solution on the basis of the given partial solution, that is, the best valid solution that contains the given partial solution. Thus, the destruction step defines a *large neighborhood*, from which a best (or nearly best) solution is determined, not by naive enumeration but by the application of a more effective alternative technique. Apart from LNS, the related literature offers algorithms that make use of alternative ways of defining large neighborhoods, such as the so-called Corridor Method [5], POPMUSIC [6], and Local Branching [7].

One of the latest algorithmic developments in the line of LNS is labelled Construct, Merge, Solve and Adapt (CMSA) [8]. Just like LNS, the main idea of CMSA is to iteratively apply a suitable exact technique to reduced problem instances, that is, sub-instances of the original problem instances. Note that the terms *reduced problem instance* and *sub-instance* refer, in this context, to a subset of the set of solutions to the tackled problem instance which is obtained by a reduction of the search space. The idea of both algorithms—LNS and CMSA—is to identify substantially reduced sub-instances of a given problem instance such that the sub-instances contain high-quality solutions to the original problem instance. This might allow the application, for example, of an exact technique with reasonable computational effort to the sub-instance in order to obtain a high-quality solution to the original problem instance. In other words, both algorithms employ techniques for reducing the search space of the tackled problem instances.

1.1 Our Contribution

Although both LNS and CMSA are based on the same general idea, the way in which the search space is reduced differs from one to the other. Based on this difference we had the intuition that LNS would (generally) work better than CMSA for problems for which solutions are rather large, and the opposite would be the case in the context of problems for which solutions are rather small. The size of solutions is hereby measured by the number of solution components (in comparison to the total number) of which they are composed. For example, in the case of the travelling salesman problem, the complete set of solution components is composed of the edges of the input graph. Moreover, solutions consist of exactly n components, where n is the number of vertices of the input graph. The above-mentioned intuition is based on the consideration that, for ending up in some high-quality solution, LNS needs to find a path of over-lapping solutions from the starting solution to the mentioned high-quality solution. The smaller the solutions are, the more difficult it should be to find such a path. A theoretical validation of our intuition seems, a priori, rather difficult to achieve. Therefore, we decided to study empirical evidence that would support (or refute) our intuition. For this purpose, we used the multi-dimensional knapsack problem (MDKP). As will be outlined later, for this problem it is possible to generate both, problem instances for which solutions are small and problem instances for

which solutions are large. We implemented both LNS and CMSA for the MDKP and performed an empirical study of the results of both algorithms for problem instances over the whole range between small and large solutions. The outcome of the presented study is empirical evidence for the validity of our intuition.

1.2 Outline of the Paper

The remainder of this paper is organized as follows. Section 2 provides a general, problem-independent, description of both LNS and CMSA, whereas Section 3 describes the application of both algorithms to the MDKP. The empirical study in the context of the MDKP is presented in Section 4, and the conclusions and an outline of future work is given in Section 5.

2 General Description of the Algorithms

In the following we provide a general description of both LNS and CMSA in the context of problems for which the exact technique used to solve sub-instances is a general-purpose integer linear programming (ILP) solver. For the following discussion we assume that a problem instance I is characterized by a complete set C of solution components. In the case of the well-known travelling salesman problem, for example, C consists of all edges of the input graph. Moreover, solutions are represented as subsets of C. Finally, any sub-instance in the context of CMSA—denoted by C'—is also a subset of C. Solutions to C' may only be formed by solution components from C'.

2.1 Large Neighborhood Search

The pseudo-code of a general ILP-based LNS is provided in Algorithm 1. First, in line 2 of Algorithm 1, an initial incumbent solution S_{cur} is generated in function GenerateInitialSolution($\mathcal{I}$). Solution S_{cur} is then partially destroyed at each iteration, depending on the *destruction rate* D_r. The way in which the incumbent solution is destroyed (randomly versus heuristically) is a relevant design decision. The resulting partial solution S_{partial} is fed to the ILP solver; see function ApplyILPSolver(S_{partial}, t_{max}) in line 7 of Algorithm 1. Apart from S_{partial}, this function receives a time limit t_{max} as input. Note that the complete solver is forced to include S_{partial} in any considered solution. This means that the corresponding sub-instance comprises all solutions that contain S_{partial}. As a result, the complete solver provides the best valid solution found within the available computation time t_{max}. This solution, denoted by S'_{opt}, may or may not be the optimal solution to the tackled sub-instance. This depends on the given computation time limit t_{max} for each application of the complete solver. Finally, in the LNS version used in this paper, the better solution between S'_{opt} and S_{cur} is carried over to the next iteration. This seems, at first sight, restrictive. In particular, other—more probabilistic—ways of selecting between S'_{opt} and S_{cur} would be possible. However, in turn the algorithm is equipped with a variable

Algorithm 1 Large Neighborhood Search (LNS)

1: **input:** problem instance $\mathcal{I}$, values for parameters D^l, D^u, D^{inc}, and $t_{\max}$
2: $S_{\text{cur}} :=$ GenerateInitialSolution($\mathcal{I}$)
3: $S_{\text{bsf}} := S_{\text{cur}}$
4: $D_r := D^l$
5: **while** CPU time limit not reached **do**
6: $S_{\text{partial}} :=$ DestroyPartially(S_{cur}, D_r)
7: $S'_{\text{opt}} :=$ ApplyILPSolver(S_{partial}, $t_{\max}$)
8: **if** S'_{opt} is better than S_{bsf} **then** $S_{\text{bsf}} := S'_{\text{opt}}$
9: **if** S'_{opt} is better than S_{cur} **then**
10: $S_{\text{cur}} := S'_{\text{opt}}$
11: $D_r := D^l$
12: **else**
13: $D_r := D_r + D^{\text{inc}}$
14: **if** $D_r > D^u$ **then** $D_r := D^l$
15: **end if**
16: **end while**
17: **return** S_{bsf}

destruction rate D_r, which may vary between a lower bound D^l and an upper bound D^u. Hereby, D^l and D^u are parameters of the algorithm. A proper setting of these parameters enables the algorithm to escape from local minima. Note that the adaptation of D_r is managed in the style of variable neighborhood search algorithms [9]. In particular, if S'_{opt} is better than S_{cur}, the value of D_r is set back to the lower bound D^l. Otherwise, the value of D_r is incremented by D^{inc}, which is also a parameter of the algorithm. If the value of D_r—after this update—exceeds the upper bound D^u, it is set back to the lower bound D^l.

2.2 Construct, Merge, Solve and Adapt

The pseudo-code of an ILP-based CMSA algorithm is provided in Algorithm 2. Each algorithm iteration consists of the following actions. First, the best-so-far solution S_{bsf} is initialized to $\emptyset$, indicating that no such solution exists yet. Moreover, the restricted problem instance C', which is—as mentioned before—a subset of the complete set C of solutions components, is initialized to the empty set. Then, at each iteration, the restricted problem instance C' is augmented in the following way (see lines 5 to 11): n_{a} solutions are probabilistically generated in function ProbabilisticSolutionGeneration(C). The components found in the constructed solutions are added to C'. Hereby, the so-called age of each of these solution components ($age[c]$) is set to zero. Once C' was augmented in this way, a complete solver is applied in function ApplyILPSolver(C') to find a possibly optimal solution S'_{opt} to the restricted problem instance C'. If S'_{opt} is better than the current best-so-far solution S_{bsf}, solution S'_{opt} is taken as the new best-so-far solution. Next, sub-instance C' is adapted on the basis of solution S'_{opt} in conjunction with the age values of the solution components; see

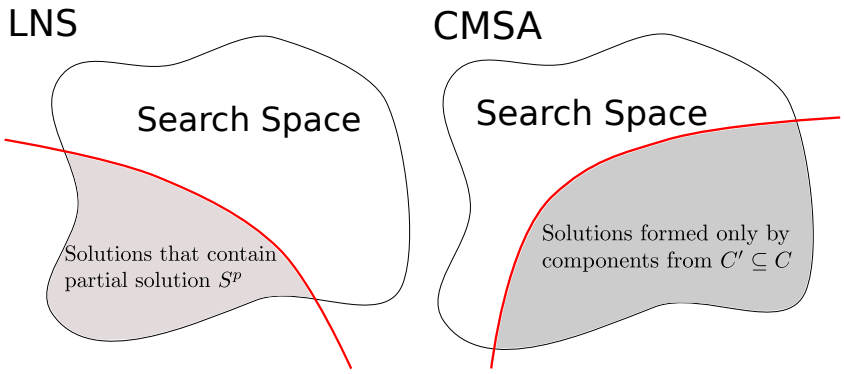
Algorithm 2 Construct, Merge, Solve and Adapt (CMSA)

1: **input:** problem instance $\mathcal{I}$, values for parameters n_{a}, $age_{\max}$, and $t_{\max}$
2: $S_{\mathrm{bsf}} := \emptyset$; $C' := \emptyset$
3: $age[c] := 0$ for all $c \in C$
4: **while** CPU time limit not reached **do**
5: **for** $i := 1, \ldots, n_{\mathrm{a}}$ **do**
6: $S :=$ ProbabilisticSolutionGeneration(C)
7: **for** all $c \in S$ **and** $c \notin C'$ **do**
8: $age[c] := 0$
9: $C' := C' \cup \{c\}$
10: **end for**
11: **end for**
12: $S'_{\mathrm{opt}} :=$ ApplyILPSolver$(C', t_{\max})$
13: **if** S'_{opt} is better than S_{bsf} **then** $S_{\mathrm{bsf}} := S'_{\mathrm{opt}}$
14: Adapt$(C', S'_{\mathrm{opt}}, age_{\max})$
15: **end while**
16: **return** s_{bsf}

function Adapt$(C', S'_{\mathrm{opt}}, age_{\max})$ in line 14. This is done as follows. First, the age of each solution component in $C' \setminus S'_{\mathrm{opt}}$ is incremented while the age of each solution component in $S'_{\mathrm{opt}} \subseteq C'$ is re-initialized to zero. Subsequently, those solution components from C' with an age value greater than $age_{\max}$—which is a parameter of the algorithm—are removed from C'. This causes that solution components that never appear in solutions derived by the complete solver do not slow down the solver in subsequent iterations. On the other side, components which appear in the solutions returned by the complete solver should be maintained in C'.

2.3 Search Space Reduction in LNS and CMSA

The way in which the search space of the tackled problem instance is reduced by LNS, respectively CMSA, can be summarized as follows. LNS keeps an incumbent solution which, at each iteration, is partially destroyed. This results in a partial solution. The reduced search space consists of all solutions to the original problem instance that contain this partial solution. This is graphically illustrated in Figure 1a. CMSA, on the other side, reduces the search space as follows: at each iteration, solutions to the original problem instance are constructed in a probabilistic way, using a greedy function as bias. The solution components found in these solutions are joined, forming a subset C' of the complete set of solution components. The set of solutions to the original problem instance that can be generated on the basis of the components in C' form the reduced search space in CMSA. This is graphically presented in Figure 1b.



(a) Search space reduction in LNS (b) Search space reduction in CMSA

Fig. 1: The way in which the search space is reduced in LNS, respectively CMSA. The term *search space* refers to the set of all valid solutions to the tackled problem instance. The grey-colored sub-spaces indicate the search spaces of the tackled sub-instances.

3 Application to the MDKP

For the aim of finding empirical evidence for the intuition phrased in the introduction of this work, we make use of the so-called *multi-dimensional knapsack problem* (MDKP), a well studied NP-hard combinatorial optimization problem and a popular test case for new algorithmic proposals (see, for example, [10–12]). The reason for choosing the MDKP is that it is parametrizable, as we will outline in more detail below.

The MDKP is defined as follows. Given is a set $C=\{1,\ldots,n\}$ of n items, and a set $K=\{1,\ldots,m\}$ of m different resources. Each resource $k \in K$ is available in a certain quantity (*capacity*) $c_k > 0$, and each item $i \in C$ requires from each resource $k \in K$ a given amount $r_{i,k} \geq 0$ (*resource consumption*). Moreover, each item $i \in C$ has associated a profit $p_i > 0$. Note that, in the context of the MDKP, the set C of items corresponds to the complete set of solution components.

A feasible solution to the MDKP is a selection (subsets) of items $S \subseteq C$ such that for each resource k the total consumption over all selected items $\sum_{i \in S} r_{i,k}$ does not exceed the resource's capacity c_k. The objective is to find a feasible item selection S of maximum total profit $\sum_{i \in S} p_i$. The MDKP can be stated in terms of an ILP as follows:

$$\text{maximize} \quad \sum_{i \in C} p_i \cdot x_i \tag{1}$$

$$\text{s.t.} \quad \sum_{i \in C} r_{i,k} \cdot x_i \leq c_k \qquad \forall k \in K \tag{2}$$

$$x_i \in \{0,1\} \qquad \forall i \in C \tag{3}$$

Hereby, inequalities (2) limit the total consumption for each resource and are called *knapsack constraints*.

For the following discussion keep in mind that when referring to valid solutions, we mean solutions that are valid and, at the same time, *non-extensible*. A valid solution S is called non-extensible, if no $i \in C \setminus S$ can be added to S without destroying its property of being a valid solution. The reasons for choosing this problem for our study is, as mentioned above, that it is highly parametrizable. With this we refer to the fact that problem instances in which the capacities of the resources are rather high are characterized by rather large valid solutions containing many items. The opposite is the case when resource capacities are low. This means that the MDKP permits to generate problem instances over the whole range of sizes of valid solutions.

3.1 Solving the Sub-instances to Optimality

For solving a sub-instance determined by a partial solution S_{partial} in the context of LNS to optimality, the following constraints must be added to the ILP model for the MDKP that was outlined above:

$$x_i = 1 \quad \forall i \in S_{\text{partial}} \tag{4}$$

Similarly, for solving a sub-instance C' in the context of CMSA to optimality we simply have to apply the ILP model using set C' instead of the complete set C.

3.2 Constructing Solutions for the MDKP

Apart from solving the sub-instances to optimality, we require a way for generating the initial solution in the case of LNS and for generating solutions at each iteration of CMSA in a probabilistic way. For both purposes we used the greedy heuristic outlined in the following. Henceforth it is assumed that the items in C are ordered w.r.t. the following *utility values* in a non-increasing way:

$$u_i \leftarrow \frac{p_i}{\sum_{k \in K} r_{i,k}/c_k} \qquad \forall\, i \in C. \tag{5}$$

That is, the items in C are ordered such that $u_1 \geq u_2 \geq \ldots \geq u_n$. This means that an item $i \in C$ has position/index i due to its utility value. The utility values are used as a static greedy weighting function in the heuristic described in Algorithm 3. This heuristic simply adds items in the order determined by the utility values to an initially empty partial solution S until no further item fits w.r.t. the remaining resource capacities.

The probabilistic way of constructing a solution employed in CMSA (function ProbabilisticSolutionGeneration(C) in line 6 of Algorithm 2) also adds one item at a time until no further item can be added without violating the constraints.

Algorithm 3 Greedy Heuristic for the MDKP

1: **input:** a MDKP instance $\mathcal{I}$
2: $S \leftarrow \emptyset$
3: **for** $i \leftarrow 1, \ldots, n$ **do**
4: **if** $\left(\sum_{j \in S} r_{j,k}\right) + r_{i,k} \leq c_k,\ \forall k = 1, \ldots, m$ **then**
5: $S \leftarrow S \cup \{i\}$
6: **end if**
7: **end for**
8: **return** S

At each solution construction step, let S denote the current partial solution and let l denote (the index of) the last item added to S. Remember that item l has index l. In case $S = \emptyset$, let $l = -1$. In order to choose the next item to be added to S, the first up to l_{size} items starting from item $l + 1$ that fit w.r.t. all resources are collected in a set L. Hereby, L is commonly called the *candidate list* and l_{size}, which is an important parameter, is called the *candidate list size*. In order to choose an item from L, a number $\nu \in [0, 1)$ is chosen uniformly at random. In case $\nu \leq d_{\text{rate}}$, the item $i^* \leftarrow \min\{i \in L\}$ is chosen and added to S. Otherwise—that is, in case $\nu > d_{\text{rate}}$—an item i^* from L is chosen uniformly at random. Just like l_{size}, the *determinism rate* d_{rate} is an input parameter of the algorithm for which a well-working value must be found.

3.3 Partial Destruction of Solutions in LNS

The last algorithmic aspect that must be specified is the way in which solutions in LNS are partially destroyed. Two variants were considered. In both variants, given the incumbent solution S, $\max\{3, \lfloor D_r \cdot |S| \rfloor\}$ items are chosen at random and are then deleted from S. However, while this choice is made uniformly at random in the first variant, the greedy function outlined above is used in the second variant in an inverse-proportional way in order to bias the random choice of items to be deleted. However, as we were not able to detect any benefit from the biased random choice, we decided to use the first variant for the final experimental evaluation.

4 Empirical Study

Both, LNS and CMSA, were coded in ANSI C++ using GCC 4.7.3 for compilation. The experimental evaluation was performed on a cluster of computers with "Intel® Xeon® CPU 5670" CPUs of 12 nuclei of 2933 MHz and (in total) 32 Gigabytes of RAM. Moreover, all ILPs in LNS and CMSA were solved with IBM ILOG CPLEX V12.1 (single-threaded mode).

In the following we describe the set of benchmark instances generated to test the two algorithms. Then, we describe the tuning experiments in order to determine a proper setting for the parameters of LNS and CMSA. Finally, the experimental results are presented.

4.1 Problem instances

The following set of benchmark instances was created using the methodology described in [13, 10]. In particular, we generated benchmark instances of $n \in \{100, 500, 1000, 5000, 10000\}$ items and $m \in \{10, 30, 50\}$ resources. The so-called *tightness* of problem instances refers hereby to the size of the capacities. In the way of generating instances that we used—first described in [13, 10]—the tightness of an instance can be specified by means of a parameter α which may take values between zero and one. The lower the value of α, the tighter is the resulting problem instance and the smaller are the solutions to the respective problem instance. In order to generate instances over the whole range of tightness values we chose $\alpha \in \{0.1, 0.2, \ldots, 0.8, 0.9\}$. More specifically, for our experiments we generated 30 random instances for each combination of values of the three above-mentioned parameters (n, m and α). For all instances, the resource requirements $r_{i,j}$ were chosen uniformly at random from $\{1, \ldots, 1000\}$. In total, the generated benchmark set consist of 4050 problem instances.

4.2 Tuning

We made use of the automatic configuration tool irace [14] for both algorithms. irace was applied for each combination of n (number of items) and α (the tightness value). More specifically, for each combination of n and α we generated three random instances for each $m \in \{10, 30, 50\}$, that is, in total nine tuning instances were generated for each application of irace. The budget of irace was set to 1000. Moreover, the following computation time limits were chosen for both LNS and CMSA: 60 CPU seconds for instances with $n = 100$, 120 CPU seconds for those with $n = 500$, 210 CPU seconds for those with $n = 1000$, 360 CPU seconds for those with $n = 5000$, and 600 CPU seconds for those with $n = 10000$.

Parameters of CMSA. The important parameters of CMSA that are considered for tuning are the following ones: (1) the number of solution constructions per iteration (n_{a}), (2) the maximum allowed age ($age_{\max}$) of solution components, (3) the determinism rate (d_{rate}), (4) the candidate list size (l_{size}), and (5) the maximum time in seconds allowed for CPLEX per application to each sub-instance ($t_{\max}$). The following parameter value ranges were chosen concerning the five parameters of CMSA.

- $n_{\text{a}} \in \{10, 30, 50\}$
- $age_{\max} \in \{1, 5, 10, \text{inf}\}$, where inf means that no solution component is ever removed from the sub-instance.
- $d_{\text{rate}} \in \{0.0, 0.3, 0.5, 0.7, 0.9\}$, where a value of 0.0 means that the selection of the next solution component to be added to the partial solution under construction is always done randomly from the candidate list, while a value of 0.9 means that solution constructions are nearly deterministic.
- $l_{\text{size}} \in \{3, 5, 10\}$
- $t_{\max} \in \{1.0, 2.0, 4.0, 8.0\}$ (in CPU seconds) for all instances with $n \in \{100, 500\}$, and $t_{\max} \in \{2.0, 4.0, 8.0, 16.0, 32.0\}$ for all larger instances.

Parameters of LNS. The parameters of LNS considered for tuning are the following ones: (1) the lower and upper bounds—that is, D^l and D^u—of the destruction rate, (2) the increment of the destruction rate (D^{inc}) , and (3) the maximum time $t_{\max}$ (in seconds) allowed for CPLEX per application to a sub-instance. The following parameter value ranges were chosen concerning the five parameters of CMSA.

- $(D^l, D^u) \in \{(0.1, 0.1), (0.2, 0.2), (0.3, 0.3), (0.4, 0.4), (0.5, 0.5), (0.6, 0.6), (0.7, 0.7), (0.8, 0.8), (0.9, 0.9), (0.1, 0.3), (0.1, 0.5), (0.3, 0.5), (0.3, 0.7), (0.3, 0.9), (0.1, 0.9)\}$. Note that when $D^l = D^u$, the destruction rate D_r is fixed.
- $D^{\text{inc}} \in \{0.01, 0.02, \ldots, 0.08, 0.09\}$
- The value range for $t_{\max}$ was chosen in the same way as for CMSA (see above).

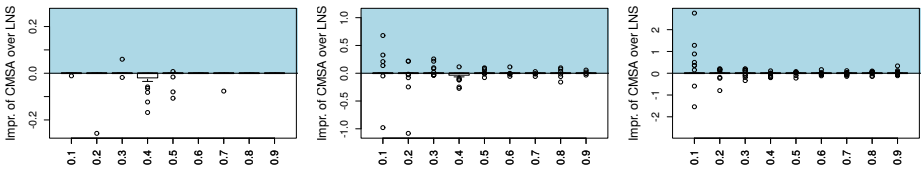
The results of the tuning processes are shown in the three sub-tables of Figure 4 in Appendix A.

4.3 Results

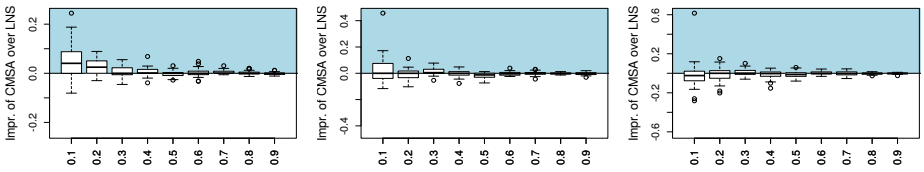
Both LNS and CMSA were applied to all problem instances exactly once, with the computation time limits as outlined at the beginning of Section 4.2. The results are shown graphically by means of boxplots in Figure 2. Note that there is one graphic per combination of n (the number of items) and m (the number of resources). The x-axis of each graphic ranges from the 30 instances of tightness value $\alpha = 0.1$ to the 30 instances of tightness value $\alpha = 0.9$, that is, from left to right we move from instances with small solutions—that is, solutions containing few components—to instances with large solutions—that is, solutions containing many components. The boxes in these boxplots show the improvement of CMSA over LNS (in percent). This means that when data points have a positive sign (that is, greater than zero), CMSA has obtained a better result than LNS, and vice versa. In order to improve the readability of these figures, the area of data points with positive signs has a shaded background.

The following main observation can be made: In accordance with our intuition that CMSA should have advantages over LNS in the context of problems with small solutions, it can be observed that CMSA generally has advantages over LNS when the tightness values of instances are rather small. This becomes more and more clear with growing instance size (n) and with a decreasing number of resources (m). In turn, LNS generally has advantages over CMSA for instances with a high tightness value, that is, for instances with large solutions.

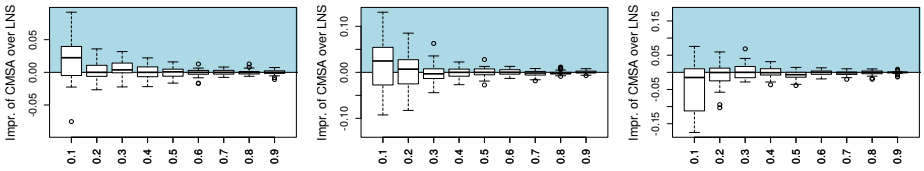
In order to shed some further light on the differences between CMSA and LNS, we also measured the percentage of items—that is, solution components—that appeared in at least one of the solutions visited by the algorithm within the allowed computation time. This information is provided in the graphics of Figure 3 by means of barplots. Again, we present one graphic per combination of n and m. The following observations can be made:



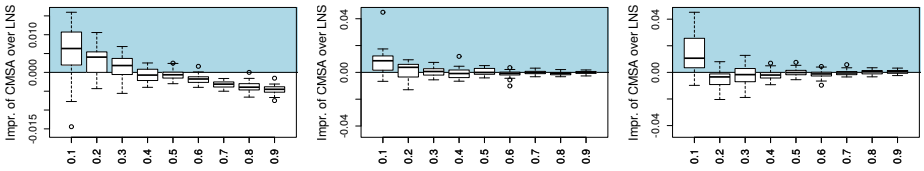
(a) $n = 100$, $m = 10$. (b) $n = 100$, $m = 30$. (c) $n = 100$, $m = 50$.



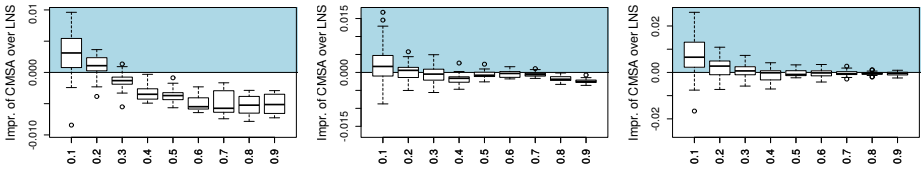
(d) $n = 500$, $m = 10$. (e) $n = 500$, $m = 30$. (f) $n = 500$, $m = 50$.



(g) $n = 1000$, $m = 10$. (h) $n = 1000$, $m = 30$. (i) $n = 1000$, $m = 50$.



(j) $n = 5000$, $m = 10$. (k) $n = 5000$, $m = 30$. (l) $n = 5000$, $m = 50$.



(m) $n = 10000$, $m = 10$. (n) $n = 10000$, $m = 30$. (o) $n = 10000$, $m = 50$.

Fig. 2: Improvement of CMSA over LNS (in percent). Each box shows the differences for the corresponding 30 instances. Note that negative values indicate that LNS obtained a better result than CMSA, and vice versa.

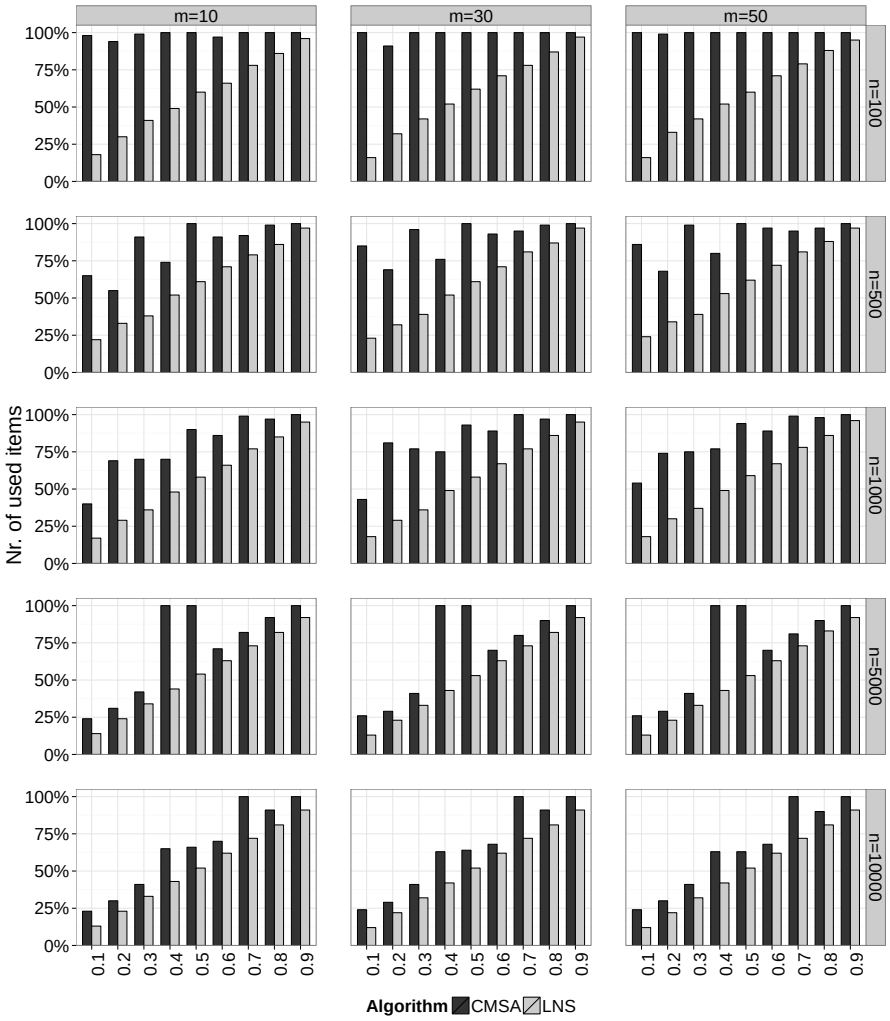


Fig. 3: Percentage of the items—that is, solution components—that were used in at least one visited solution. Each bar shows the average over the respective 30 problem instances.

– First of all, the percentage of used items is always much higher for CMSA than for LNS. This means that, with the optimized parameter setting as determined by irace, CMSA is much more *explorative* than LNS. This, apparently, pays off in the context of problems with small solutions. On the downside, this seems rather not beneficial when problems are characterized by large solutions.
– Additionally, it can be observed that the difference in the percentage of the usage of items between CMSA and LNS decreases with growing instances size (n). This can be explained by the fact that the absolute size of the sub-instances that are generated by CMSA naturally grows with growing problem size, and—as the efficiency of CPLEX for solving these sub-instances decreases with growing sub-instance size—the parameter values as determined by irace are such that the relative size of the sub-instances is smaller for large problem instances, which essentially means that the algorithm is less explorative.

5 Conclusions and Future Work

In this work we have given first empirical evidence that supports our initial intuition that LNS should generally work better than CMSA for problems in which solutions contain rather many solution components, and vice versa. This has been shown by means of experimental results in the context of the multi-dimensional knapsack problem. In the near future we intent to confirm this empirical evidence by the application to additional optimization problems.

Finally, we would like to clarify the following aspect. Our intuition obviously only holds for problems for which, a priori, neither LNS nor CMSA have advantages over the other one. In fact, it is not very difficult to find problems for which CMSA generally has advantages over LNS, no matter if solutions are small or large. Consider, for example, problems for which the number of variables and/or constraints in the respective ILP model are so large that the problem cannot be solved simply because of memory restrictions. This is the case in ILP models in which the number of variables and/or constraints are super-linear concerning the input parameters of the problem. Due to its specific way of reducing the search space, CMSA tackles sub-instances that correspond to reduced ILP models. This is not the case of LNS. Even though parts of the solution are fixed, the complete original ILP model must be built in order to solve the corresponding sub-instance. Therefore, CMSA can be applied in these cases, while LNS cannot be applied. An example of such a problem is the repetition-free longest common subsequence problem [15]. Contrarily, it is neither difficult to think about problems for which LNS generally has advantages over CMSA. Consider, for example, a problem where the main difficulty is not the size of ILP model but rather the computational complexity. Moreover, let us assume that when fixing a part of the solution, the sub-instance becomes rather easy to be solved, which is—for example—the case in problems with strong symmetries. In such a case LNS will most probably have advantages over CMSA. An example of such a problem is the most strings with few bad columns problem [16].

References

1. Talbi, E., ed.: Hybrid Metaheuristics. Volume 434 of Studies in Computational Intelligence. Springer (2013)
2. Blum, C., Raidl, G.R.: Hybrid Metaheuristics – Powerful Tools for Optimization. Springer (2016)
3. Boschetti, M.A., Maniezzo, V., Roffilli, M., Bolufé Röhler, A.: Matheuristics: Optimization, simulation and control. In Blesa, M.J., Blum, C., Di Gaspero, L., Roli, A., Sampels, M., Schaerf, A., eds.: Proceedings of HM 2009 – 6th International Workshop on Hybrid Metaheuristics. Volume 5818 of Lecture Notes in Computer Science., Springer Berlin Heidelberg (2009) 171–177
4. Pisinger, D., Ropke, S.: Large Neighborhood Search. In Gendreau, M., Potvin, J.Y., eds.: Handbook of Metaheuristics. Volume 146 of International Series in Operations Research & Management Science. Springer US (2010) 399–419
5. Caserta, M., Voß, S.: A corridor method based hybrid algorithm for redundancy allocation. J. Heuristics **22**(4) (2016) 405–429
6. Lalla-Ruiz, E., Voß, S.: POPMUSIC as a matheuristic for the berth allocation problem. Ann. Math. Artif. Intell. **76**(1-2) (2016) 173–189
7. Fischetti, M., Lodi, A.: Local branching. Mathematical Programming **98**(1) (2003) 23–47
8. Blum, C., Pinacho, P., López-Ibáñez, M., Lozano, J.A.: Construct, merge, solve & adapt: A new general algorithm for combinatorial optimization. Computers & Operations Research **68** (2016) 75–88
9. Hansen, P., Mladenović, N.: Variable Neighborhood Search: Principles and Applications. European Journal of Operational Research **130**(3) (2001) 449–467
10. Chu, P.C., Beasley, J.E.: A genetic algorithm for the multidimensional knapsack problem. Discrete Applied Mathematics **49**(1) (1994) 189–212
11. Leung, S., Zhang, D., Zhou, C., Wu, T.: A hybrid simulated annealing metaheuristic algorithm for the two-dimensional knapsack problem. Computers and Operations Research **39**(1) (2012) 64–73
12. Kong, X., Gao, L., Ouyang, H., Li, S.: Solving large-scale multidimensional knapsack problems with a new binary harmony search algorithm. Computers and Operations Research **63** (2015) 7–22
13. Hanafi, S., Freville, A.: An efficient tabu search approach for the 0-1 multidimensional knapsack problem. European Journal of Operational Research **106**(2-3) (1998) 659–675
14. López-Ibáñez, M., Dubois-Lacoste, J., Pérez Cáceres, L., Birattari, M., Stützle, T.: The irace package: Iterated racing for automatic algorithm configuration. Operations Research Perspectives **3** (2016) 43–58
15. Blum, C., Blesa, M.J.: Construct, merge, solve and adapt: Application to the repetition-free longest common subsequence problem. In Chicano, F., Hu, B., García-Sánchez, P., eds.: Proceedings of EvoCOP 2016 – 16th European Conference on Evolutionary Computation in Combinatorial Optimization. Number 9595 in Lecture Notes in Computer Science, Springer International Publishing (2016) 46–57
16. Lizárraga, E., Blesa, M.J., Blum, C., Raidl, G.R.: Large neighborhood search for the most strings with few bad columns problem. Soft Computing (2016) In press.

Appendix A: Tuning results

α	$n = 100$					$n = 500$					$n = 1000$				
	n_{a}	$age_{\max}$	d_{rate}	l_{size}	$t_{\max}$	n_{a}	$age_{\max}$	d_{rate}	l_{size}	$t_{\max}$	n_{a}	$age_{\max}$	d_{rate}	l_{size}	$t_{\max}$
0.1	30	1	0.3	5	1.0	50	1	0.3	3	4.0	10	1	0.9	10	2.0
0.2	30	1	0.5	3	2.0	50	10	0.9	5	1.0	50	1	0.5	5	8.0
0.3	30	10	0.7	5	8.0	30	1	0	3	4.0	30	1	0.5	3	4.0
0.4	30	10	0.7	5	8.0	10	10	0.9	5	2.0	50	1	0.7	3	8.0
0.5	50	1	0.7	5	2.0	10	1	0.5	3	4.0	10	1	0.5	3	8.0
0.6	30	5	0.7	3	8.0	30	1	0.9	5	2.0	10	5	0.9	5	4.0
0.7	10	5	0.7	3	4.0	30	5	0.9	3	4.0	30	5	0.7	3	8.0
0.8	50	1	0.9	3	4.0	30	1	0.9	3	8.0	30	5	0.9	3	4.0
0.9	10	1	0.3	3	8.0	10	1	0.5	10	2.0	10	1	0.3	10	2.0

(a) Tuning CMSA for instances with $n \in \{100, 500, 1000\}$.

α	$n = 5000$					$n = 10000$				
	n_{a}	$age_{\max}$	d_{rate}	l_{size}	$t_{\max}$	n_{a}	$age_{\max}$	d_{rate}	l_{size}	$t_{\max}$
0.1	10	1	0.5	3	4.0	30	1	0.7	5	8.0
0.2	10	1	0.9	5	8.0	50	1	0.9	5	8.0
0.3	10	1	0.9	5	8.0	50	5	0.9	5	16.0
0.4	10	1	0.5	10	4.0	10	inf	0.5	3	16.0
0.5	10	1	0	10	8.0	50	10	0.9	5	32.0
0.6	50	inf	0.9	3	16.0	50	inf	0.9	3	32.0
0.7	30	1	0.9	3	16.0	10	1	0.7	10	32.0
0.8	10	5	0.9	3	32.0	30	inf	0.9	3	32.0
0.9	10	1	0	10	16.0	10	10	0.9	3	32.0

(b) Tuning CMSA for instances with $n \in \{5000, 10000\}$.

α	$n = 100$				$n = 500$				$n = 1000$				$n = 5000$				$n = 10000$			
	D^l	D^u	D^{inc}	$t_{\max}$	D^l	D^u	D^{inc}	$t_{\max}$	D^l	D^u	D^{inc}	$t_{\max}$	D^l	D^u	D^{inc}	$t_{\max}$	D^l	D^u	D^{inc}	$t_{\max}$
0.1	0.8	0.8	0.08	4.0	0.9	0.9	0.05	1.0	0.8	0.8	0.06	4.0	0.8	0.8	0.04	8.0	0.9	0.9	0.08	8.0
0.2	0.9	0.9	0.06	4.0	0.9	0.9	0.03	1.0	0.9	0.9	0.06	2.0	0.9	0.9	0.03	16.0	0.9	0.9	0.06	32.0
0.3	0.9	0.9	0.03	1.0	0.8	0.8	0.06	4.0	0.9	0.9	0.07	8.0	0.7	0.7	0.01	16.0	0.8	0.8	0.06	8.0
0.4	0.9	0.9	0.03	4.0	0.7	0.7	0.03	1.0	0.8	0.8	0.01	2.0	0.9	0.9	0.08	8.0	0.9	0.9	0.02	8.0
0.5	0.9	0.9	0.01	2.0	0.6	0.6	0.05	1.0	0.8	0.8	0.08	2.0	0.7	0.7	0.05	4.0	0.9	0.9	0.01	32.0
0.6	0.9	0.9	0.01	2.0	0.7	0.7	0.06	1.0	0.8	0.8	0.03	4.0	0.9	0.9	0.02	4.0	0.8	0.8	0.03	32.0
0.7	0.9	0.9	0.05	2.0	0.6	0.6	0.05	1.0	0.9	0.9	0.02	2.0	0.9	0.9	0.06	8.0	0.9	0.9	0.08	16.0
0.8	0.8	0.8	0.08	4.0	0.9	0.9	0.02	2.0	0.8	0.8	0.07	4.0	0.8	0.8	0.03	8.0	0.9	0.9	0.01	16.0
0.9	0.1	0.9	0.09	1.0	0.8	0.8	0.05	1.0	0.9	0.9	0.03	2.0	0.8	0.8	0.05	8.0	0.8	0.8	0.02	16.0

(c) Tuning LNS.

Fig. 4: Tuning results.